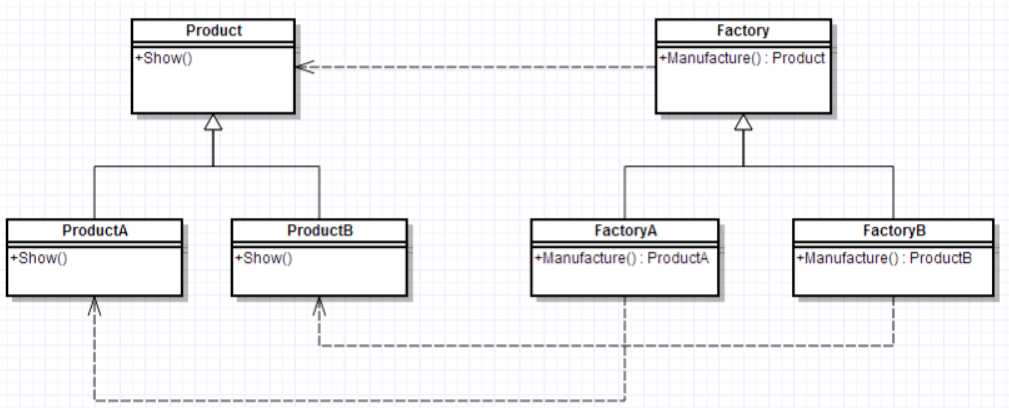
1.Factory method:

1）Definition

The factory method pattern, also known as the factory pattern, polymorphic factory pattern, and virtual constructor pattern, is responsible for defining the common interface for creating objects by defining the factory parent class, while the subclass is responsible for generating concrete objects.

2）Main function

Deferring the instantiation of the class (the creation of the specific product) to a subclass of the factory class (the specific factory), which determines which class should be instantiated (created).

  
abstract class Factory{

public abstract Product Manufacture();

}

abstract class Product{

public abstract void Show();

}

class ProductA extends Product{

@Override

public void Show() {

System.out.println("生产出了产品A");

}

}

class ProductB extends Product{

@Override

public void Show() {

System.out.println("生产出了产品B");

}

class FactoryA extends Factory{

@Override

public Product Manufacture() {

return new ProductA();

class FactoryB extends Factory{

@Override

public Product Manufacture() {

return new ProductB();}}

public class FactoryPattern {

public static void main(String[] args){

FactoryA mFactoryA = new FactoryA();

mFactoryA.Manufacture().Show();

FactoryB mFactoryB = new FactoryB();

mFactoryB.Manufacture().Show();}}

Advantages：

1. It follows the open-close principle
2. Consistent with the single responsibility principle
3. Instead of using the static factory approach, you can have a hierarchical structure based on inheritance.

Disadvantages：

When adding new products, in addition to adding new product classes, the corresponding specific factory classes should also be provided. The number of system classes will be increased in pairs, which increases the complexity of the system to some extent. At the same time, there are more classes to compile and run, which brings some extra overhead to the system.

Considering the extensibility of the system, it is necessary to introduce an abstract layer, which is defined in the client code, which increases the abstraction and difficulty of understanding of the system. In addition, DOM, reflection and other technologies may be used in the implementation, which increases the difficulty of the system.

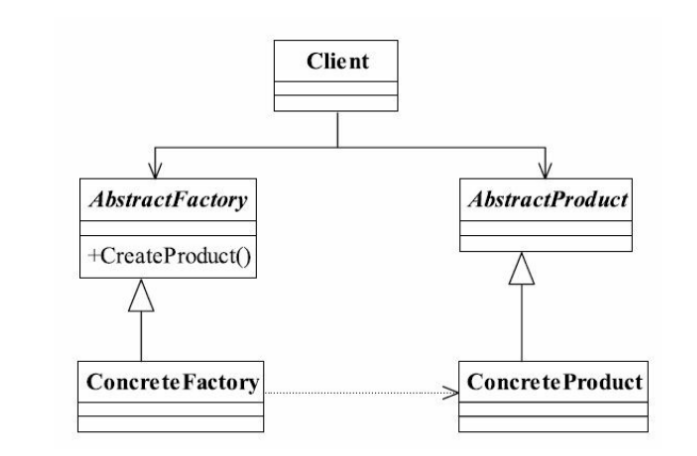
While the factory method is guaranteed to be closed for modifications, for classes that use the factory method, you still need to modify the instantiated specific factory class if you want to change another product.

A specific factory can only create one specific product

2.Abstract Factory Pattern

1）definition：

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



Advantages

1) encapsulation. The implementation class of each product is not the concern of the high-level module, it is the concern of the interface, is the abstraction, it does not care how the object is created, it is the responsibility of the factory class, as long as I know who the factory class is, I can create a required object, save time and effort.

2) the constraints within the product family are non-public.

Disadvantages

The biggest drawback of the abstract factory pattern is that product family extensions are difficult. If we want to add A product C, that is, increase the product family from the original A and B to three, we first need to add the createProductC() method to the abstract class AbstractCreator, and then modify both implementation classes

代码：

public abstract class AbstractProductA {

public void shareMethod() {

}

public abstract void doSomething();

}

public class productA1 extends AbstractProductA {

public abstract void doSomething(){

System.out.println("产品A1的实现方法");

}

}

public class productA2 extends AbstractProductA {

public abstract void doSomething(){

System.out.println("产品A2的实现方法");

}

}

public abstract class AbstractCreator {

public abstract AbstracProductA createProductA();

public abstract AbstracProductB createProductB();

}

public class Creator1 extends AbstractCreator {

public AbstracProductA createProductA(){

return new ProductA1(); }

public AbstracProductB createProductB() {

return new ProductB1();}}

public class Creator2 extends AbstractCreator {

public AbstracProductA createProductA(){

return new ProductA2();

}

public AbstracProductB createProductB() {

return new ProductB2();

}

3.bridge pattern

Definition

The bridge pattern is to separate the abstract part from its implementation part so that they can all change independently.

Advantages:

1. Separation of abstraction and implementation.

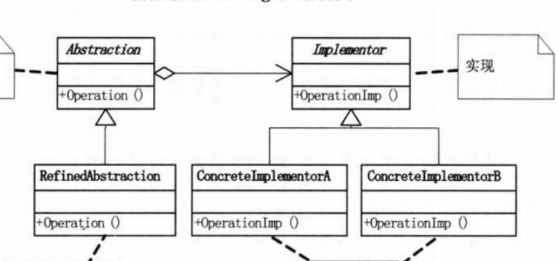
2. Excellent expansion ability.

3. Implementation details are transparent to customers.

Disadvantages:

1. Increased the difficulty of system understanding and design. Since the aggregation relation is built on the abstraction layer, developers are required to design and program the abstraction.

2. It is required to correctly identify two dimensions with independent changes in the system, so its scope of use has certain limitations.



public interface Implementor {

void doSomething();

void doAnything();

}

public class ConcreteImplementor1 implements Implementor {

@Override

public void doSomething() {

}

@Override

public void doAnything() { } }

public class ConcreteImplementor2 implements Implementor {

@Override

public void doSomething() {

}

@Override

public void doAnything() {

}

}

public abstract class Abstraction {

private Implementor imp;

public Abstraction(Implementor imp) {

this.imp = imp;

}

public void request() {

this.imp.doSomething();

}

public Implementor getImp() {

return imp;

}

public class RefinedAbstraction extends Abstraction {

public RefinedAbstraction(Implementor imp) {

super(imp);

}

@Override

public void request() {

super.request();

super.getImp().doAnything();

}

public class Client {

public static void main(String[] args) {

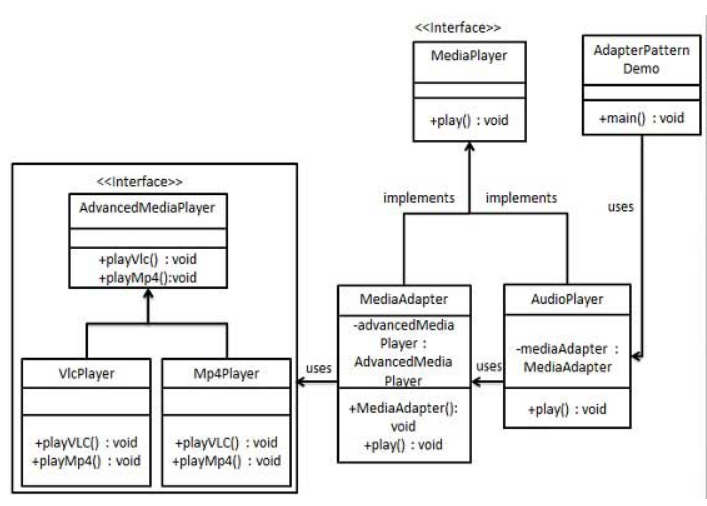
Implementor imp = new ConcreteImplementor1();

Abstraction abs = new RefinedAbstraction(imp);

abs.request();

4.Adapter pattern

Definition: to convert the interface of one class into another interface that the customer wants. The adapter pattern allows classes that would otherwise not work together because of interface incompatibilities to work together.



Advantages:

1. You can run any two unrelated classes together. 2. Improved class reuse. 3. Increased class transparency. 4. Flexibility.

Disadvantages:

1, too much use of adapters, will make the system very messy, not easy to grasp the whole. For example, it is obvious that what is called is the interface of A, but in fact, it is internally adapted to be the implementation of interface b. if too many such situations occur in A system, it is nothing short of A disaster. So if it's not necessary, you can simply refactor the system without using the adapter. 2. Since JAVA inherits at most one class, at most one adapter class can be adapted, and the target class must be abstract.

代码：

public interface MediaPlayer {

public void play(String audioType, String fileName);

}

public interface AdvancedMediaPlayer {

public void playVlc(String fileName);

public void playMp4(String fileName);

}

public class VlcPlayer implements AdvancedMediaPlayer{

public void playVlc(String fileName) {

System.out.println("Playing vlc file. Name: "+ fileName);

}

public void playMp4(String fileName) { } }

public class Mp4Player implements AdvancedMediaPlayer{

public void playVlc(String fileName) { }

public void playMp4(String fileName) {

System.out.println("Playing mp4 file. Name: "+ fileName);

}

public class MediaAdapter implements MediaPlayer {

AdvancedMediaPlayer advancedMusicPlayer;

public MediaAdapter(String audioType){

if(audioType.equalsIgnoreCase("vlc") ){

advancedMusicPlayer = new VlcPlayer();

} else if (audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer = new Mp4Player();

}

}

public void play(String audioType, String fileName) {

if(audioType.equalsIgnoreCase("vlc")){

advancedMusicPlayer.playVlc(fileName);

}else if(audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer.playMp4(fileName);

}

}

}

public class AudioPlayer implements MediaPlayer {

MediaAdapter mediaAdapter;

public void play(String audioType, String fileName) {

if(audioType.equalsIgnoreCase("mp3")){

System.out.println("Playing mp3 file. Name: "+ fileName);

} else if(audioType.equalsIgnoreCase("vlc")

|| audioType.equalsIgnoreCase("mp4")){

mediaAdapter = new MediaAdapter(audioType);

mediaAdapter.play(audioType, fileName);

}

else{

System.out.println("Invalid media. "+

audioType + " format not supported");

}

}

}

public class AdapterPatternDemo {

public static void main(String[] args) {

AudioPlayer audioPlayer = new AudioPlayer();

audioPlayer.play("mp3", "beyond the horizon.mp3");

audioPlayer.play("mp4", "alone.mp4");

audioPlayer.play("vlc", "far far away.vlc");

audioPlayer.play("avi", "mind me.avi");

}

}

**5.build pattern**

Definition：The abstract interface specified for creating the individual parts of a Product object

Advantages：

1) build a complex object step by step. A Product is usually made up of several different child objects, and the builder provides the interface to build and get the final Product object.

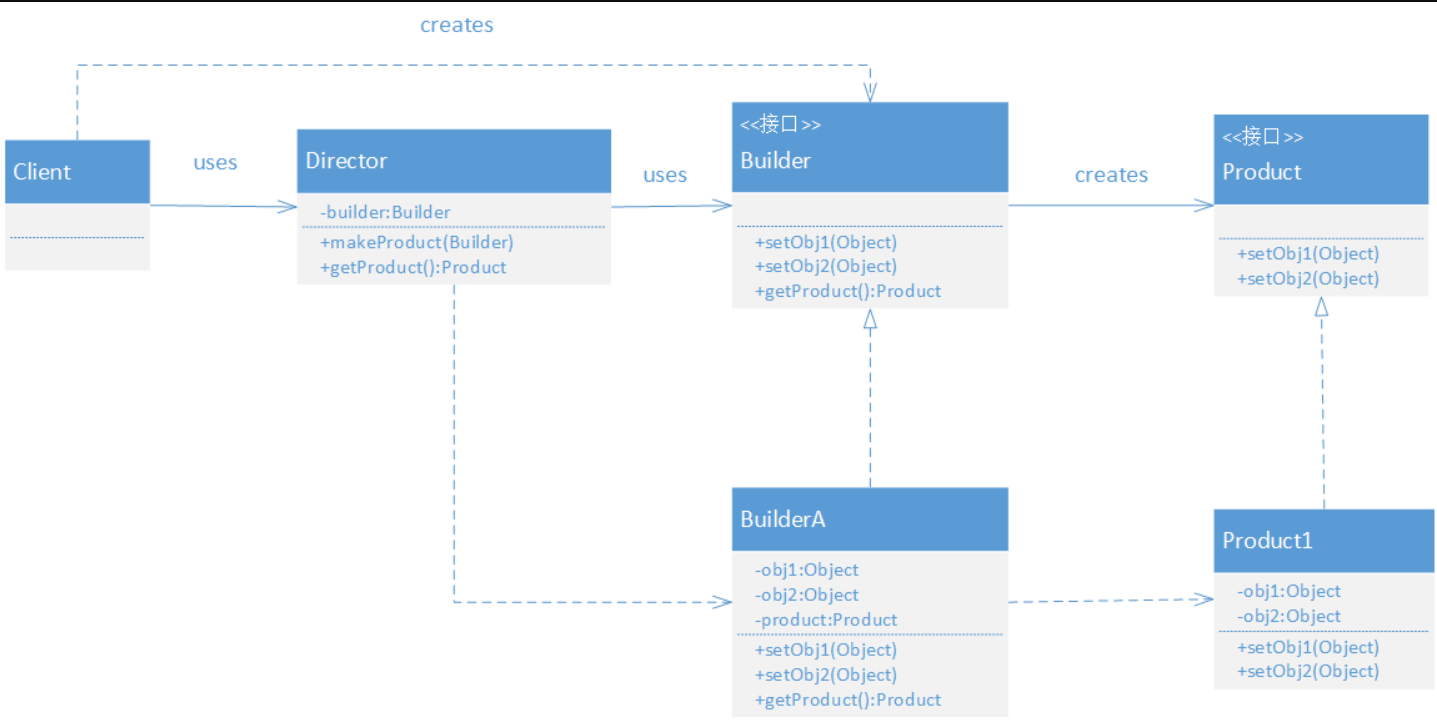
2) support for multiple build representations. The Builder pattern encapsulates the build process, allowing users to create different representations by simply passing in the parameters they care about and the parameters they need, rather than the complex build process.

3) decouple the creation of components and assembly mode of components, and the type of components and assembly mode of components shall not affect each other

Disadvantages：

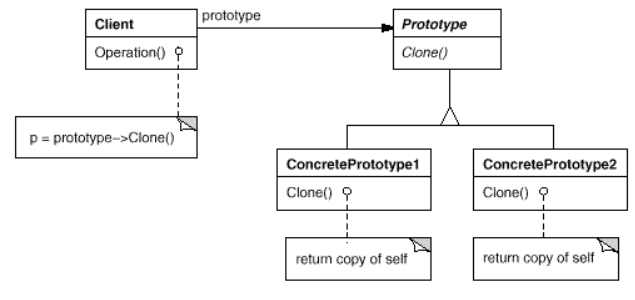
1) it is not suitable to create products containing similar subobjects, such as composite mode;

2) when Product adds components, it may lead to a large number of modifications of Builder, its subclasses and Director;



6. Prototype

Definition：The prototype pattern is usually used to create a new object, then we use this object to complete some operations on the object, we can quickly create an object through the prototype pattern



Advantages:

(1) according to the requirements of the client, the object is created dynamically. The client does not need to know the creation details of the object to facilitate the maintenance and extension of the code.

(2) creating an Object using the prototype pattern is much better than creating a new Object because the clone method of the Object class is a local method that operates directly on the binary stream in memory, especially when copying large objects. So consider using the prototype pattern when you need to repeatedly create similar objects.

(3) clearer and simpler code structure.